부분키 기법과 압축 기법을 혼용한 주기억장치 상주형 다차원 색인 구조

(A Main Memory-resident Multi-dimensional Index Structure Employing Partial-key and Compression Schemes)

심 정 민 † 민 영 수 †† 송 석 일 ††† 유 재 수 ††††
(Jeong Min Shim) (Young Soo Min) (Seok Il Song) (Jae Soo Yoo)

요 약 최근 중앙처리장치와 주기억장치간의 병목 현상에 의한 성능 저하를 극복하기 위해 캐시를 고려한 색인 구조들이 제안되었다. 이런 색인 구조들의 궁극적인 목표는 엔트리 크기를 줄여 팬-아웃(fan-out)을 증가시키고, 캐시 접근 실패를 최소화하여 시스템의 성능을 높이는 것이다. 엔트리의 크기를 줄이는 기법에 따라 기존의 색인 구조들을 두 가지로 구분할 수 있다. 하나는 좌표 값을 고정된 비트로 양자화 함으로써, MBR 키를 압축하는 것이다. 또 다른 하나는 MBR들의 각 좌표 값 중에 그들의 부모 MBR과 같지 않은 좌표 값만을 저장하는 것이다. 우선, 본 논문에서는 두 기법의 특성들을 적절히 조합한 새로운 색인 구조를 제안하고, 기존에 제시된 두 접근법을 따르는 주기억장치 상주형 다차원 색인 구조를 다양한 환경에서 성능 평가한다. 또한, 기존의 색인 구조와 비교를 통해 제안하는 색인 구조의 우수성을 보인다.

키워드 : 캐시, 다차원 색인구조, 주기억장치 상주형 데이타베이스 시스템

Abstract Recently, to relieve the performance degradation caused by the bottleneck between CPU and main memory, cache conscious multi-dimensional index structures have been proposed. The ultimate goal of them is to reduce the space for entries so as to widen index trees and minimize the number of cache misses. The existing index structures can be classified into two approaches according to their entry reduction methods. One approach is to compress MBR keys by quantizing coordinate values to the fixed number of bits. The other approach is to store only the sides of minimum bounding regions (MBRs) that are different from their parents partially. In this paper, we propose a new index structure that exploits the properties of the both techniques. Then, we investigate the existing multi-dimensional index structures for main memory database system through experiments under the various work loads. We perform various experiments to show that our approach outperforms others.

Key words : cache, multi-dimensional index structures, main-memory database systems

1. 서 론

컴퓨터 시스템의 주기억장치 가격이 점점 하락함에 따라, 주기억장치 상주형 데이타베이스관리시스템이 다양한 응용분야에 보급되고 있다. 주기억장치 상주형 데이타베이스관리시스템이 디스크 기반의 데이타베이스관리시스템보다 10배 이상의 성능을 보인다는 것은 이미 잘 알려진 사실이다. 전통적인 디스크 기반의 데이타베이스관리시스템에서 병목현상의 주된 요인은 디스크 I/O였다. 트랜잭션의 실행 시간에서 가장 큰 부분을 차지하는 디스크 I/O를 줄이기 위한 많은 연구들이 수행되었으며 그 결과로, 디스크 I/O로 인한 성능 저하 현상은 상당히 감소되었다.

유사하게 최근 중앙처리장치와 주기억장치의 성능 차이가 점점 커짐에 따라 캐시 실패가 데이타베이스관리시스템의 새로운 병목현상으로 대두되었다. 특히, 주기억장치 상주형 데이타베이스관리시스템에서는 L2 캐시

· 본 연구는 한국과학재단 목적기초연구(R01-2003-000-10627-0)와 KISTEP 생물정보학사업 지원으로 수행되었음
† 학생회원 : 한국전자통신연구원 디지털홈연구단 연구원
jmshim@etri.re.kr
†† 비 회 원 : 충북대학교 정보통신공학과
minys@netdb.chungbuk.ac.kr
††† 비 회 원 : 충주대학교 컴퓨터공학과 교수
sisong@chungju.ac.kr
†††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@cbucc.chungbuk.ac.kr
논문접수 : 2003년 9월 9일
심사완료 : 2004년 3월 30일

실패 회수를 줄이는 것이 성능 향상에 가장 큰 요인으로 부각되었다[1,2]. 이에 따라, L2 캐시 실패를 줄여 주기억장치 상주형 데이타베이스관리시스템의 성능을 향상시키려는 연구들이 활발히 진행되었다[3-8]. 1990년대 말 이후로 캐시를 고려한 색인 구조는 주기억장치 상주형 데이타베이스관리시스템의 성능을 향상시키는데 초점이 맞추어졌다. 또한, 2000년대 초에는 지리정보시스템(Geographical Information System : GIS), 위치기반시스템(Location Based System : LBS)과 같은 응용분야가 주기억장치 상주형 데이타베이스시스템을 기반으로 구축되야 할 필요성이 제기됨에 따라서 캐시 실패를 고려한 다차원 색인 구조가 제안되기에 이르렀다.

캐시를 고려한 대표적인 다차원 색인 구조들은 CR-트리[7], PR-트리[8] 그리고 노드의 크기를 캐시 라인 또는 그 정수 배의 크기로 사용하는 R-트리[9]가 있다. CR-트리는 각 MBR의 좌표 값들을 부모 MBR의 각 축의 작은 값을 기준으로 하여 상대적으로 표현하고, 이 상대적인 좌표 값을 고정된 개수의 비트로 양자화하여 MBR 키 값을 압축한다. PR-트리는 부모 MBR의 좌표 값과 다른 좌표 값만을 부분적으로 저장한다. 이 색인 구조들의 궁극적인 목표는 엔트리 크기 감소를 통해 팬-아웃을 증가시켜 캐시 접근 실패를 최소화하는 것이다.

본 논문에서는 CR-트리와 PR-트리의 특성들을 조합한 새로운 색인 구조를 제안한다. 그리고, 노드의 크기를 캐시 라인 또는 그 정수 배로 구성하여 CR-트리, PR-트리, R-트리와 같은 기존 색인 구조들의 성능을 실험을 통해 비교한다. MBR 압축 기법을 사용하는 CR-트리는 압축 레벨(4비트~8비트)에 따라 R-트리보다 4배에서 8배정도 많은 엔트리들을 저장한다. 그렇지만, 부분 MBR 기법을 사용하는PR-트리는 엔트리 수가 3~7개정도로 적을 때 좋은 성능을 보인다. 이런 이유로 인하여 부분 MBR 기법을 CR-트리에 적용하는 것은 적절하지 않아 보일 수 있다. 본 논문에서는 PR-트리와 CR-트리의 특징들을 이용한 새로운 색인구조를 연구한다. 그리고, 제안하는 색인 구조를 다양한 환경에서 성능평가를 수행하여 CR-트리, PR-트리, R-트리 보다 우수함을 보인다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 기존의 캐시를 고려한 일차원 색인 구조와 다차원 색인 구조에 대하여 논의한다. 3장에서는 CR-트리와 PR-트리를 자세히 분석하고, 두 기법을 적절하게 조합한 PCR-트리를 제안한다. 4장에서는 다양한 실험을 통해 제안하는 색인 구조의 우수성을 보이고, 마지막으로 5장에서 결론을 맺는다.

2. 캐시를 고려한 기존 색인 구조

1990년대 말에 Rao와 Ross는 주기억장치 상주형 데이타베이스 관리 시스템의 성능을 향상시키기 위해서 캐시를 고려한 두 개의 색인 구조를 제안했다. CSS-트리(Cache-Sensitive Search-트리)[3]와 CSB+-트리(Cache-Sensitive B+-트리)[4]가 바로 그것이다. 조사한 바에 따르면, 이들 두 색인구조는 캐시를 고려한 색인 구조의 발단이다. OLAP 환경을 위해 고려된 CSS-트리는 엔트리들의 밀집도가 매우 높은 공간 효율적인 B+-트리이다. 색인구조에서 자식노드를 가리키는 포인터를 제거하고 물리적으로 연속적인 공간에 키를 저장한다. 노드의 자식은 k-ary 방법을 이용하여 쉽게 찾을 수 있다. CSB+-트리는 효율적인 변경을 지원해야 하는 OLTP 환경에 대한 색인 구조에 CSS-트리의 기법을 적용하였다. CSB+-트리는 부모노드에 있는 포인터를 줄이기 위해 자식 노드들을 물리적으로 인접한 공간에 저장한다.

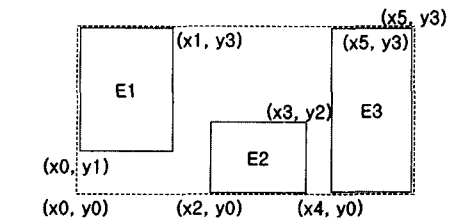
반면에, [5]의 부분 키 기법은 전체 키 중에서 일부의 고정된 크기만을 저장한다. pkB+-트리는 고정된 크기의 일부 부분 키만을 사용함으로서, 키 비교 비용과 캐시 접근 실패를 줄였다. pB+-트리는 효과적으로 캐시 접근 실패를 줄이기 위해 프리페치 기법을 B+-트리에 도입했다[6]. 검색을 하는 동안 발생하는 캐시 접근 실패를 줄이기 위해 노드의 크기를 캐시 라인의 정수 배로 확장하였다. 따라서, 색인 구조의 높이는 낮아진다. 또한, 색인 구조를 순회하는 동안 발생하는 캐시 접근 실패를 줄이기 위해 점프-포인터(jump-pointer) 기법을 제안했다. 부분 키 기법, 프리페치 기법과 CSS/CSB+-트리는 같은 목표를 지향하지만 접근 방법은 매우 다르다.

이상 설명한 색인 구조들은 B+-트리와 같은 일차원 색인 구조들이다. 다차원 데이타를 위한 방법들로는 CR-트리와 PR-트리가 있다. 본 논문에서는 제안하는 색인구조의 기반이 되는 이 두 색인구조의 특징을 기술한다.

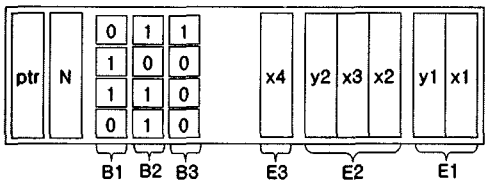
2.1 PR-트리

주기억장치 상주형 데이타베이스 관리 시스템에서 R-트리의 노드 크기는 보통 캐시 라인의 크기와 같거나 그 정수 배이다. PR-트리의 저자는 노드 크기가 작을 경우 부모 MBR의 좌표 값과 자식 MBR의 좌표 값이 어느 정도 겹치는가를 실험을 통해 알아보았다. 측정 결과, 평균적으로 한 MBR의 좌표 값 중 1.7~1.75개의 좌표 값이 겹침을 알 수 있었다.

PR-트리는 이 점을 적절히 이용한 방법이다. 그림 1은 PR-트리의 부분 MBR 기법을 설명하고 있다. 그림 1(a)에서 E1, E2, E3를 포함하는 점선은 부모 MBR이다. E1의 좌측과 위의 좌표 값 즉, x0와 y3는 부모 MBR의 좌표 값과 같다. 이것은 부모 MBR을 알고 있



(a)



(b)

그림 1 부분 MBR 기법의 예

을 경우 E1의 MBR은 2개의 좌표 값 x1과 y1만으로 표현이 가능하다. 같은 방법으로 E2는 x2, x3, y2만이 저장되어 지고, y0는 부모 MBR로부터 좌표 값을 알 수 있다. E3에 대해서는 x4만이 저장되어 지고, y0, y3, x5는 부모 MBR로부터 얻을 수 있다.

그림 1(b)는 그림 1(a)의 MBR들을 부분 MBR 기법을 적용하여 표현한 노드 구조를 보인다. PR-트리의 노드 구조는 원래 R-트리의 노드 구조와 달리 가변적이다. 2차원 데이타의 경우, 모든 MBR 키에 대해서 4개의 좌표 값을 저장하지 않고, 부모 MBR의 좌표 값과 겹치지 않는 0~4개의 좌표 값을 저장한다. ptr은 자식 노드 그룹에 대한 포인터이고, N은 노드 안의 엔트리 수이다. B1은 E1에 대한 비트-필드(bit-field)이다. 비트-필드는 4비트로 구성하고, 각각의 비트는 E1의 좌표 값들이 저장되어 있는지 여부를 나타낸다. 그림 1(b)에서 E1은 두 개의 좌표 값 x1, y1을 저장하고, B1에서 이 좌표 값들에 대응하는 두 개의 비트를 1로 설정한다. 나머지 두 개의 비트는 좌표 값이 저장되지 않았음을 표시하기 위해 0으로 설정하고, 이에 대한 좌표 값은 부모 MBR로부터 얻는다.

PR-트리를 정확하게 탐색하기 위해서는 자식 노드를 방문하기 전에 현재의 MBR을 저장해야 한다. 저장된 MBR은 방문한 자식 노드를 완전한 MBR로 복구 할 때 사용되어 진다. 자식 노드에서는 저장된 부모 MBR과 현재 노드의 MBR들을 결합함으로서 각각의 엔트리에 대해서 완전한 MBR을 얻는다.

2.2 CR-트리

부분 MBR 기법은 노드 안의 엔트리로부터 중복되는 정보를 제거함으로서 노드에 저장되는 엔트리의 수를 증가시킨다. 부분 MBR 기법을 이용하는 PR-트리는 노드의 엔트리 수를 증가시켜 노드의 팬-아웃을 증가시키는 것이 목적이다. CR-트리 또한 PR-트리와 목적은 같지만 접근 방법은 매우 다르다.

CR-트리의 기본적인 개념은 MBR 압축을 통해 팬-아웃을 증가시키는 것이다. 그림 2는 CR-트리의 MBR 압축 기법을 보여준다. 그림 2(a)는 R0~R3의 MBR들을 절대 좌표로 표현하고 있다. 그림 2(b)는 R0의 좌측-아래(150, 180)를 기준으로 하는 R1~R3의 MBR들을 상대 좌표로 표현하고 있다. 이들 상대적인 좌표들은 절대 좌표들 보다 훨씬 작은 비트로 표현이 가능하다. 하지만, 상대적인 좌표들은 부모 MBR의 크기에 영향을 받는다. 다시 말해서, 부모 MBR이 매우 크다면 상대적인 좌표로 표현된 MBR의 좌표 값도 커진다. 그림 2(c)는 16레벨 또는 4비트로 양자화된 R1~R3의 좌표 값들을 보인다. 즉, R0의 좌표 값을 16개로 나누고, R0에 포함되어 있는 R1~R3의 좌표 값을 0에서 15까지의 수로 표현한다. 이렇게 하면 부모 MBR이 커지더라도 상대적인 좌표들을 고정된 비트로 표현 할 수 있다. 이렇게 만들어진 MBR을 QRMBR(quantized relative representation of MBR)이라고 부른다. 그림 2(c)에서처럼 QRMBR들은 양자화로 인하여 원래의 MBR보다 커

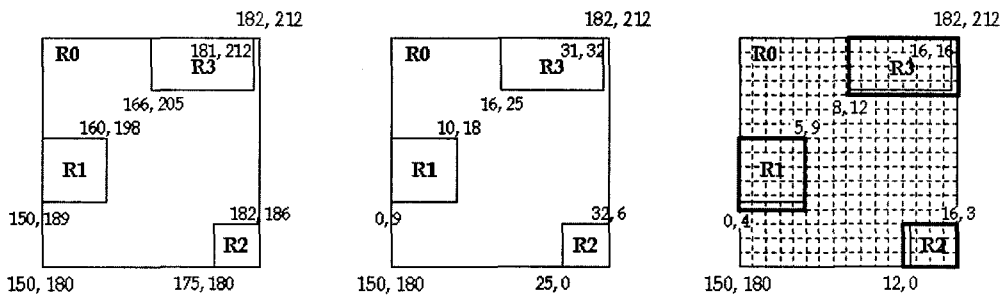


그림 2 MBR 압축 기법의 예

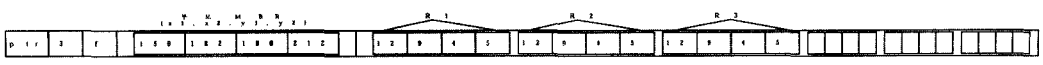


그림 3 CR-트리의 노드 구조

질 수 있다.

CR-트리는 색인 키로서 QRMBR을 사용하는 R-트리이다. MBR을 간단하게 하는 양자화의 레벨은 모든 노드에서 같다. 그림 3은 3개의 엔트리를 포함하는 CR-트리의 노드 구조를 나타낸다. 저장된 엔트리들이 단말 노드인지 비 단말 노드인지를 나타내는 플래그, 엔트리 수 그리고 엔트리의 모든 자식 MBR을 포함하는 최소 크기의 참조 MBR을 유지한다. 참조 MBR은 노드에 저장된 QRMBR을 계산하기 위하여 사용된다. 비 단말 노드들은 (ptr, n, f, refMBR, QRMBR) 형태의 엔트리 정보를 저장한다. ptr은 자식 노드의 주소이고, QRMBR은 자식 노드 MBR의 상대적인 좌표를 양자화 한 값이다. 단말 노드들도 (ptr, n, f, refMBR, QRMBR) 형태의 엔트리 정보를 저장하는데, ptr은 객체의 주소를 나타내고 QRMBR은 객체 MBR의 상대적인 좌표를 양자화 한 것이다.

3. PCR-트리 : 제안하는 캐시를 고려한 다차원 색인 구조

이 절에서는 먼저 PR-트리와 CR-트리의 성능개선측면을 자세히 분석하여 제안하는 색인 구조의 접근 방향을 설정한다. 또한 도출한 접근 방향에 따라 설계한 캐시를 고려한 다차원 색인 구조의 특징 및 알고리즘을 기술한다.

3.1 PR-트리와 CR-트리의 분석

PR-트리의 부분 MBR 기법은 엔트리의 수가 3~7개 정도로 비교적 적을 때 좋은 성능을 얻어낼 수 있다. 이 기법에서는 노드의 엔트리 수가 증가함에 따라 부모 MBR과 겹치는 좌표의 비율은 줄어든다. 예를 들어, 2차원 데이타 공간에서 최대 엔트리 수가 3일 경우 부모 MBR과 겹치는 최소 좌표 수는 4개이다. 만약 하나의 좌표 값이 4바이트를 차지한다면, 3개의 엔트리에 대한 공간은 48바이트만큼 필요하다. 이 경우에 겹치는 비율은 33%가 되고, 32바이트의 저장 공간이 사용된다. PR-트리는 좌표 값이 부모 MBR과 겹치는지 여부를 표현하기 위해 하나의 좌표 값마다 1비트의 추가적인 정보가 필요하다. 2차원의 MBR에 대해서는 4비트가 요구된다. 따라서 3개의 엔트리에 대한 비트-필드로서 12비트가 필요하고, 3개의 엔트리를 표현하기 위한 공간의 총계는 34.5바이트가 된다. 그렇지만, 노드의 최대 엔트리 수가 40개 일 경우에도 겹치는 최소 좌표 수는 4개이다. 결과적으로, 엔트리를 표현하기 위한 공간은

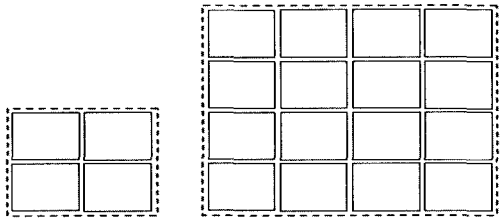


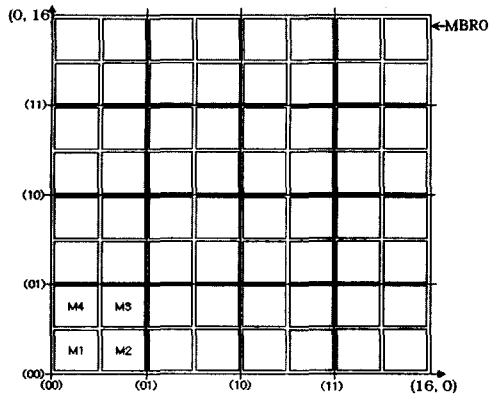
그림 4 같은 크기의 MBR을 각각 4, 16개 포함하는 두 노드

(640-16)바이트 + (4*40)비트 = 644바이트가 된다. 이렇게 되면 공간적인 효율을 거의 얻을 수 없다.

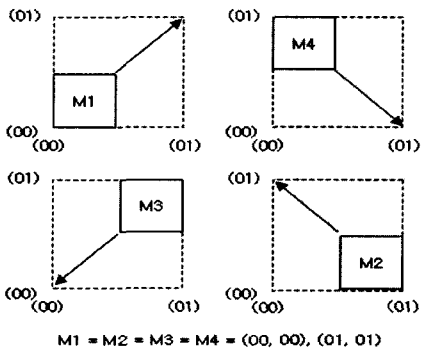
그림 4(a)의 점선으로 표현된 부모 MBR에 같은 크기의 MBR 4개가 균등하게 분배되어 있다. 이 그림에서 노드에 포함된 좌표들은 총 16개이고, 그 중에 8개는 부모 MBR과 겹친다. 이 경우 50%의 저장공간이 절약된다. 반면에, 그림 4(b)에서 노드에 포함된 좌표들은 총 64개이고, 그 중에 16개의 좌표가 부모 MBR과 겹친다. 이것은 전체 좌표의 25%에 해당한다. 부모 MBR의 좌표와 겹치는 좌표들은 노드의 엔트리 수가 증가함에 따라 감소한다.

CR-트리의 MBR 압축 기법은 MBR 정보의 크기를 감소시켜 노드의 팬-아웃을 증가시키는 것이다. 캐시 라인의 크기가 64바이트이고 하나의 좌표가 4비트를 차지한다면, 하나의 엔트리는 2바이트로 표현이 가능하고 QRMBR이외의 정보들이 25바이트를 차지하므로 노드의 최대 엔트리 수는 19개이다. 1장에서 언급한 것처럼, CR-트리에 부분 MBR 기법을 적용하는 것은 의미가 없어 보인다. 그렇지만, CR-트리의 압축 기법은 MBR이 양자화 됨으로서 MBR을 확장한다. 따라서, 노드의 부모 MBR과 겹치는 MBR의 좌표의 개수가 증가하게 될 것이라고 예측할 수 있다. 그림 5는 MBR을 압축한 후 겹치는 좌표가 증가하는 예를 나타낸 것이다.

그림 5(a)에서 정수(4바이트)로 표현된 MBR0는 (0, 0)에서 (16, 16)의 영역을 차지하고, 작은 사각형의 MBR들은 모두 MBR0에 포함되어 있다. 만약 2비트로 MBR을 압축한다면, MBR들은 00, 01, 10, 11과 같이 새로운 좌표 값에 따라 확장될 것이다. 이런 결과로 M1, M2, M3, M4는 같은 좌표값(00, 00), (01, 01)으로 표현된다. 유사한 방법으로, 각각의 영역에서 MBR들은 확장되어 지고, 그림 5(b)에서처럼 각 영역의 MBR들은 같은 값을 갖게 된다.



(a)



(b)

그림 5 CR-트리에서 좌표 값의 증가

그림 5(a)에서 MBR0에 포함된 MBR들의 좌표 중 12.5%가 MBR0와 겹친다. 그렇지만, MBR을 압축한 후에는 좌표의 25%가 MBR0와 겹치게 된다. 이것은 PR-트리에서 노드의 엔트리 수를 3개로 하였을 때와 유사한 결과이다. CR-트리에서 좌표의 크기를 4비트와 8비트로 압축하였을 경우 얼마나 많은 좌표들이 부모 MBR과 겹치는가를 알아보기 위해 간단한 실험을 하였다. 표 1은 실험 결과이다.

표 1 4, 8비트 압축 레벨에서 부모 MBR과 겹치는 자식 MBR의 비율

	4비트 압축	8비트 압축
균등 분포 데이타	43.75%	28.125%
실제 데이타	50%	34.375%

4비트로 압축하였을 경우 MBR의 좌표 중에 45%가 부모 MBR의 좌표 값과 중복되었다. 8비트로 하였을 경우에는 30%가 중복되었다. 이 결과에서는 추가적인 정보(각 좌표에 대한 비트-필드:1비트)는 고려하지 않았다. 추가적인 정보를 고려하더라도 4비트 압축의 경우에는 20%, 8비트 압축의 경우에는 18%의 공간 효율을 얻을 수 있다. 위의 사실로부터, 부분 MBR 기법과 MBR 압축 기법을 결합함으로서 R-트리의 성능을 향상시킬 수 있다는 사실을 이끌어 낼 수 있다.

3.2 PCR-트리의 특징

제안하는 PCR-트리의 기본적인 개념은 PR-트리의 부분 MBR 기법과 CR-트리의 MBR 압축 기법이다. 그림 6은 PCR-트리의 노드 구조를 보인다. NE(Number of Entry)는 노드가 포함하는 엔트리의 수이고, CP (Child Pointer)는 자식 노드 그룹에 대한 포인터이다. AMBR(Absolute MBR)은 노드의 엔트리들을 포함하는 MBR의 절대 좌표이고, 노드에 포함된 엔트리들의 MBR은 AMBR에 의해서 다시 계산된다. BF는 엔트리들의 비트-필드이고, ENTRIES는 AMBR의 좌표 값들과 겹치지 않는 엔트리들의 양자화 된 MBR 좌표 값이다.

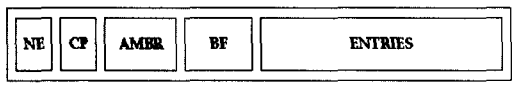


그림 6 PCR-트리의 노드 구조

노드의 엔트리 수를 나타내는 NE는 또 다른 중요한 역할을 수행한다. 모든 삽입과 삭제 연산 과정에서, 부분 MBR 기법과 MBR 압축 기법을 결합하여 노드를 구성하는 것이 MBR 압축 기법만을 사용하였을 때보다 많은 저장 공간을 차지하는지 확인한다. 만약 MBR 압축 기법만을 사용했을 때, 더 작은 저장 공간을 차지한다면 NE 값을 음수로 바꾼다. ENTRIES를 접근하기 위해서는 NE에 의해서 결정된 BF의 크기를 알아야 한다. 따라서, 부분 MBR 기법을 사용한 노드의 엔트리를 정확하게 읽기 위해서는 NE를 알아야만 한다. 그렇지만, 부분 MBR 기법이 적용되지 않은 노드에는 BF에 아무런 값도 없다. NE가 음수라는 것은 BF에 아무런 값도 갖지 않음을 나타낸다.

3.3 PCR-트리의 주요 알고리즘

3.3.1 삽입 알고리즘

PCR-트리는 기존의 캐시를 고려한 다차원 색인 구조들과 달리 두 가지의 노드 구조를 가진다. 기본적인 노드의 구조는 MBR 압축 기법과 부분 MBR 기법을 결합하여 구성하고, MBR 압축 기법만을 사용하여 구성한 노드보다 저장공간이 클 경우에는 MBR 압축 기법만을 사용한다. 따라서, 삽입할 엔트리가 각 노드를 거칠 때마다 현재 노드가 MBR 압축 기법으로만 구성되어 있는지 아닌지를 판단해야한다. 이 판단은 NE의 값이 음수인지, 양수인지를 확인하여 얻을 수 있다. 게다가, 각

레벨에서 노드가 변경될 때에는 MBR 압축 기법과 부분 MBR 기법을 결합하여 구성하고, 구성한 노드의 저장공간을 MBR 압축 기법만을 사용했을 때와 비교한다. 구성된 노드의 저장공간이 클 경우 노드를 MBR 압축 기법만을 사용하여 구성하고, NE의 값도 음수로 변경한다. 설명한 것처럼 PCR-트리는 경우에 따라 노드의 구조가 변경된다. 이처럼 엔트리를 삽입할 때마다 노드를 결정하기 위한 연산으로 인해 기존의 캐시를 고려한 다차원 색인 구조들보다 삽입 알고리즘이 복잡하다. 하지만, 이러한 연산은 공간 효율을 높여주기 때문에 노드의 분할을 최소화 시켜 분할 비용을 줄이게 하는 이점을 준다.

그림 7은 PCR-트리의 삽입 알고리즘을 나타내고 있다. PCR-트리의 삽입 알고리즘은 기존의 R-트리 계열과 유사하게 두 단계로 이루어진다. 첫 번째 단계에서는 *locateleaf()*를 통해 삽입될 엔트리가 저장되야 할 단말 노드를 찾는다. 알고리즘에서의 *locateleaf()*는 PR-트리와 CR-트리의 단말 노드 검색과 유사한 연산을 수행한다. 그렇기 때문에, 그림 7에서는 *locateleaf()*의 자세한 연산 과정을 생략하였다. *locateleaf()*에서는 단말 노드에 엔트리를 삽입한 후 상위 노드들의 연산을 처리하기 위해 방문한 노드를 *ps*에 저장한다. 두 번째 단계에서는 선택된 단말 노드가 넘침이 발생하는지 여부를 확인한다. 넘침이 발생하면 노드는 *splitnode()*에 의해 분할된다. 그렇지 않으면, 엔트리를 단말 노드에 삽입하고 단말 노드의 형태가 *pnode*인지 *qnode*인지를 결정한다. *pnode*는 부분 MBR 기법과 MBR 압축 기법을 함께 사용하여 노드를 구성하고, *qnode*는 MBR 압축 기법만을 사용하여 노드를 구성한다. 엔트리를 단말 노드에 삽입한 후 단말 노드의 AMBR이 변하였는지 확인한다. AMBR이 변하였다면 *updateMBR()*를 호출한다. *updateMBR()*은 변경된 MBR을 포함하기 위해 조상 노드의 MBR을 확장한다. *updateMBR()*에서는 변경된 MBR로 인하여 노드가 변경되어 *splitnode()*를 호출할 수 있다. *splitnode()*의 호출을 줄이기 위해 노드에 대해서 pnode와 qnode의 공간을 계산한다. 추가적인 연산으로 인해 비용이 증가할 수 있으나, 최소의 노드 분할을 하기 때문에 오히려 비용을 줄일 수 있다.

3.3.2 검색 알고리즘

PCR-트리에서 노드의 MBR은 양자화 되어있다. 검색을 수행하려는 노드가 질의 엔트리와 겹치는지를 판단하기 위해서는 노드의 MBR을 복원하거나 질의 엔트리를 현재 노드의 AMBR을 기준으로 하여 양자화 해야 한다. 현재 노드의 MBR을 복원하기 위해서는 노드가 어떤 기법을 사용했는지를 판단해야 한다. 또한, 노드의 많은 MBR을 복원하는 것보다 질의 엔트리를 양자화하

```
변수 :
  · ps : 삽입할 엔트리가 트리에서 거쳐간 노드의 경로
         를 저장하는 스택, 전역 변수
  · newentry : 노드에 삽입될 새로운 엔트리
    - 단말 노드에 삽입될 엔트리는 객체 포인터와 객체
      의 MBR을 포함
    - 비단말 노드에 삽입될 엔트리는 자식 노드의
      MBR만을 포함
  · node : 단말 노드 또는 비단말 노드
  · leaf : 단말 노드
  · pnode : 부분 MBR 기법과 MBR 압축 기법을 함께
            사용하여 구성된 노드
  · qnode : MBR 압축 기법만을 사용한 노드
  · space : 노드의 엔트리들이 저장되어 있는 실제 바이
            트 수

insert ( newentry, root )
{
   node = locateleaf( newentry, root );

   overflow = putentry( node, newentry );
   if( overflow )
   {
      splitnode( node, newentry );
      return;
   }
}

putentry ( node, newentry )
{
   newentry와 노드의 엔트리에 대한 pnode의 공간과
   qnode의 공간을 계산;

   if( space_pnode < space_qnode )
      space = space_pnode;

   if( space > maximum node size )
      return;

   node에 newentry를 삽입하고 node를 재구성;

   if( node.AMBR이newentry.MBR를 포함하지 않음)
   {
      node.AMBR를 업데이트;
      updateMBR( node.AMBR );
   }

   return false;
}

splitnode ( node, newentry )
{
   node.ENTRIES와 newentry를 두 노드 그룹으로 분류;
   두 노드 그룹에 대해서 노드의 형태를 결정(pnode 또
   는 qnode);

   newnode를 생성;
   newnode와 node에 두 노드 그룹을 저장;
}

updateMBR ( MBR )
{
   node = pop(ps);

   if( node.AMBR이 MBR을 포함 )
      return;

 MBR을 포함하도록 node.AMBR을 확장했을 때의 pnode
 와 qnode의 공간을 계산;

   if( space_pnode < space_qnode )
      space = space_pnode;

   if( space >= maximum node size )
   {
      split( node, null );
      return;
   }

     node.AMBR이 MBR을 포함하도록 업데이트;
   노드의 형태(pnode 또는 qnode)에 따라서 노드를 재구성;

   updateMBR(node.AMBR);
}
```

그림 7 Insert 알고리즘

는 것이 훨씬 간단하다. 따라서, 질의 엔트리는 검색을 수행하는 동안 각 노드에서 AMBR을 기준으로 하여 양자화한 값을 사용한다. 검색 알고리즘은 기존의 색인 구조에서 사용하는 방법과 유사하며, 삽입 알고리즘과 달리 추가적인 연산은 요구되지 않는다. PCR-트리는 기존 색인 구조들보다 노드의 팬-아웃이 증가되었기 때문에 검색 성능을 향상시킬 수 있다. 또한, 공간 효율을 증가시키기 위해 두 기법의 결합이 좋지 않을 경우에는 MBR 압축 기법만을 사용하여 노드를 구성하기 때문에 적어도 다른 색인 구조보다 성능이 뛰어난 CR-트리보다 우수한 성능을 보일 것이다.

그림 8은 검색을 수행하는 Search 알고리즘을 나타내고 있다. 제안하는 검색 알고리즘은 전반적으로 PR-트리와 CR-트리가 결합된 형태이다. 방문한 노드에 대해서 우선 노드의 형태가 *pnode*인지 *qnode*인지를 검사한다. 노드가 *qnode*이면 즉, NE의 값이 양수이면 CR-트리의 검색 알고리즘이 적용된다. 그렇지 않으면, CR-트리와 PR-트리를 결합한 형태의 검색 알고리즘을 수행하게 된다. 그림 8에서처럼 *searchnode()*에서는 우선 노드의 형태를 검사하기 위해서 엔트리의 수를 나타내는 NE를 검사한다. 양수 값의 NE는 노드가 *pnode*라는 것을 의미한다. 따라서, ENTRIES에 있는 양자화된 좌표 값들을 BF와 AMBR 값을 참조하여 모두 얻어낸다. 다음의 연산 과정은 얻어낸 좌표 값을 가지고 CR-트리의 검색 연산과정과 똑같이 수행한다.

```
searchnode ( node, range )
{
   if( node.NE >= 0 )
   {
      node.BF의 값을 검사;
      ENTRIES에서 node.BF의 값이 1인 것에 대응
      하는 값을 읽음;
   }

   나머지 연산 과정은 CR-트리와 같음;
}
```

그림 8 Search 알고리즘

3.3.3 삭제 알고리즘

PCR-트리의 삭제 알고리즘은 기존의 R-트리 계열의 색인 구조와 유사하다. 삭제 알고리즘에서도 삽입 알고리즘과 마찬가지로 노드의 형태를 결정하기 위한 계산이 필요하다. PCR-트리의 노드에 저장되어 있는 엔트리들의 정보는 양자화되어 있다. 양자화된 엔트리들은 AMBR을 이용하여 원래의 값으로 복원되어진다. 그렇기 때문에 엔트리가 삭제되더라도 AMBR이 변경되지 않는다면 계산은 필요 없다. 따라서, 삭제 연산을 수행

```
변수 :
   · delentry : 삭제할 엔트리

delete ( delentry, root )
{
   node = findleaf( root, delentry  );

   if( node == null )
      return;

   removeentry( node, delentry );

   condensetree( node );
}

removeentry ( node, delentry )
{
   node에서 delentry를 삭제;

   newAMBR = node.AMBR을 재계산;

   if( newAMBR != node.AMBR )
      node를 업데이트;
}
```

그림 9 Delete 알고리즘

할 때 삭제된 엔트리의 정보로 인해 AMBR이 변하는지 검사하고, AMBR이 변한다면 노드의 엔트리들을 다시 계산하여 노드의 형태를 결정하고 노드의 정보를 변경해야 한다.

그림 9는 삭제를 수행하는 Delete 알고리즘을 나타내고 있다. 우선 *findleaf()*를 통해 삭제할 엔트리를 찾는다. *findleaf()*에서는 기존의 R-트리 계열의 연산처럼 삭제할 엔트리를 찾지 못했을 경우에는 삭제 연산을 중단한다. *findleaf()*에서 삭제할 엔트리를 찾으면 그 엔트리를 포함하는 단말 노드를 반환하고, *removeentry()*를 호출한다. *removeentry()*에서는 반환된 단말 노드에서 *delentry*를 삭제한다. 그리고, *delentry*가 삭제되어 AMBR이 변하는지를 검사한다. AMBR이 변하였다면 노드의 엔트리 정보를 변경된 AMBR을 기준으로 재구성한다. 또한, *delentry*를 삭제하여 노드의 엔트리 정보가 매우 적어지면 단말 노드를 삭제해야 한다. 이것은 *condense()*를 통해 이루어진다. *condense()*에서는 노드의 엔트리 정보가 매우 적어 노드를 삭제하게 되면 부모 노드의 엔트리 또한 삭제해야 한다. 이 경우에도 removeentry()를 통해 AMBR을 검사하고 노드의 정보를 변경해야하는지 확인한다. 이 부분을 제외한 나머지에 대해서는 기존의 R-트리 계열과 같다.

4. 성능 평가

4.1 평가 환경

제안하는 PCR-트리의 성능을 평가하기 위해서, 기존의 주기억장치 기반의 색인 구조인 CR-트리, PR-트리, R-트리와 비교하였으며, R-트리의 노드 크기는 캐시 라인과 그 정수 배로 하였다. 성능 평가에 사용된 시스템은 펜티엄4 1GHz프로세서에 64bytes의 캐시라인 크기와 256Mbytes의 메모리를 가지며, 운영체제는 윈도우즈 2000을 사용하였다. CR-트리, PR-트리, R-트리와 제안하는 PCR-트리는 공통적으로 QS(quadratic splitting algorithm)분할 알고리즘을 사용하였다. 또한, STR(sort tile recursive)[10] 벌크로딩 알고리즘을 사용하여 벌크로딩 기법을 구현하였다.

실험에 사용된 데이타 집합은 두 가지이다. 하나는 균일하게 분포되어 있는 정수 데이타이고, 다른 하나는 실제 데이타, TIGER이다. TIGER 데이타는 웹사이트[11]에서 얻었고, 균등 분포 데이타는 다음의 과정을 거쳐 생성하였다. 우선 랜덤 함수를 이용하여 0~100사이의 정수를 산출해 내고, 그것들을 2차원 배열에 연속적으로 저장하였다. 이와 동시에 배열에서 2개의 원소들을 묶어 MBR을 반복적으로 생성하였다. MBR의 크기를 제한하지 않았기 때문에 MBR 영역의 크기는 다양하다.

STR 벌크로딩 알고리즘을 사용하여 트리를 만들고, 다양한 환경에서 검색 성능을 측정하였다. 일반적으로, 다차원 색인 구조에서 트리를 구축하는 방법으로는 준비된 데이타 집합을 한번에 삽입하는 벌크로딩과 엔트리를 하나씩 순차적으로 삽입하는 방법이 있다. 어떻게 트리를 생성하는가에 따라서 성능차이가 크다. 그래서 이 논문에서는 벌크로딩 기법으로 구축한 색인 트리뿐 아니라, 벌크로딩 기법을 사용하여 구축한 트리에 순차적으로 엔트리들을 삽입한 후에도 검색 성능을 측정하였다. 추가적으로 삽입된 엔트리들은 다양한 크기와 위치를 갖는다. 또한, 벌크로딩 기법으로 구축한 트리에 10,000개의 엔트리를 삽입하면서 삽입 성능을 측정하였다. 검색 성능 평가를 위해 10,000개의 범위 질의를 사용하였으며, 데이타 전체 영역의 0.001%~1%까지의 다양한 범위 질의에 대해서 접근하는 노드 수, 검색 시간 그리고 캐시 접근 실패 수를 측정했다. 실제 그래프를 그릴 때는 이들의 평균값을 이용하였다.

4.2 실험 결과

그림 10~12은 균등 분포 데이타에 대해서 접근하는 노드 수, 검색 시간 그리고 캐시 접근 실패 수의 결과를 보인다. CR-트리(4)와 PCR-트리(4)는 트리에서 각각의 MBR 좌표들을 4비트로 압축한다. 즉, 색인 구조의 MBR 키가 2바이트임을 의미한다. 반면에 CR-트리(8)와 PCR-트리(8)는 각각의 MBR 좌표들이 8비트로 압축되어 색인 구조의 MBR 키가 4바이트가 된다. 기존의 색인 구조들 중에서 CR-트리는 R-트리와 PR-트리보다 우수한 성능을 보인다. 그리고, 3.3.2절의 검색 알고리즘에서 언급한 것처럼 PCR-트리의 검색 성능은 적어도 CR-트리보다 좋은 성능을 보인다. 따라서, PCR-트리는 기존 색인 구조들보다 우수하다는 것을 알 수 있다. 그림 10~12에서 PCR-트리는 다른 색인 구조들보다 우수한 성능을 보인다. 그림 10은 각각의 색인 구조에 대해 검색을 수행하였을 때 접근하는 노드의 수를 보인다. 노드 크기가 증가할수록 트리의 높이는 낮아지기 때문에 모든 트리는 노드 크기가 증가할수록 노드 접근 수가 감소한다. 그림 10에서 PCR-트리(8)는 나머지 트리들보다 뛰어난 성능을 보인다. 그러나, 4비트로 압축을 하였을 때는 PCR-트리가 CR-트리(8)보다 성능이 떨어짐을 볼 수 있다. [7]에서 언급한 것처럼, 압축 레벨은 색인 구조의 성능에 영향을 준다. 그림 10에서 PCR-트리(4)가 CR-트리(8)보다 노드 접근 수가 많은 것은 압축 레벨을 4비트로 할 경우 노드에 저장되는 엔트리는 증가하지만 MBR 압축 기법의 특성상 확장영역이 커짐에 따라 불필요한 노드를 접근하는 수가 많아지기 때문이다. 그림 11~12에서도 PCR-트리(8)는 다른 트리들보다

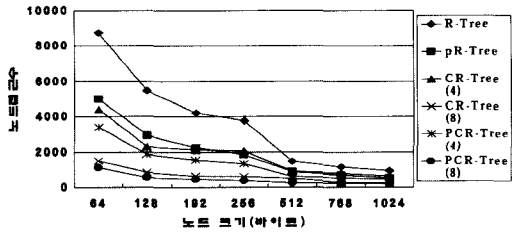


그림 10 노드 접근 수 (균등 분포 데이타)

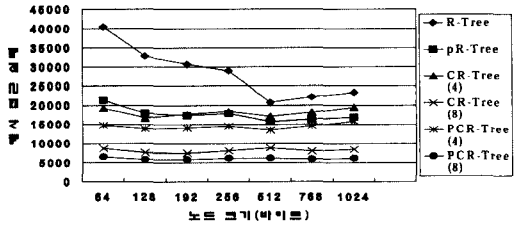


그림 11 캐시 접근 실패 수 (균등 분포 데이타)

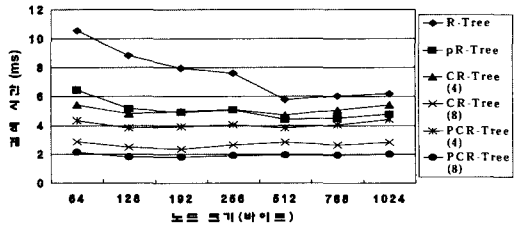


그림 12 검색 시간 (균등 분포 데이타)

성능이 뛰어나다. 그리고, 제안하는 PCR-트리는 R-트리와 PR-트리와는 달리 노드 크기가 변하더라도 변화의 폭이 거의 변하지 않는다.

그림 13~15는 실제(TIGER) 데이타에 대한 검색 성능 결과이다. 이 결과는 그림 10~12의 결과와 다르다. 그림 13에서 접근하는 노드 수는 노드의 크기가 증가함에 따라 감소하는 것을 볼 수 있다. 그림 13에서 접근하는 노드 수의 변화 폭은 그림 10에서보다 적으며, 노드 크기가 1024바이트일 경우에는 접근 노드 수가 대부분의 색인 구조에서 유사하다. 그림 14~15에서 PCR-트리(4)는 CR-트리(8)보다 성능이 떨어지지만, PCR-트리(8)는 나머지 트리들보다 우수함을 볼 수 있다. 그림 14~15에서는 균등 분포 데이타와 달리 노드 크기가 증가함에 따라 캐시 접근 실패 수와 검색시간이 증가하는 것을 볼 수 있다. 그림 14에서는 노드 크기가 256바이트일 때부터 노드 크기가 증가함에 따라 캐시 접근 실패가 증가하고, 그림 15에서는 노드 크기가 증가함에 따라, 검색 시간이 증가한다. 그림 13에서 노드 크기가 증가함에 따라 접근하는 노드 수가 감소하는 폭은 크지 않다. 노드 크기가 증가하게 되면 트리의 높이가 낮아지고 접근하는 노드 수는 줄어들지만, 노드 크기가 증가하면 하나의 노드를 읽기 위해 여러 번의 캐시 접근 실패를 초래하게 된다. 그렇기 때문에 노드 크기가 증가하더라도 감소폭이 적다면 캐시 접근 실패 수는 감소하지 않고 증가할 수 있다. 이것은 그림 14~15에서 노드 크기가 증가하더라도 성능이 떨어지는 원인이 된다.

PCR-트리와 CR-트리는 고정된 비트로 양자화를 하기 때문에 부모 MBR 영역의 크기가 커지면 MBR 영역이 더 확장된다. 따라서, 균등 분포 데이타보다 전체 영역이 넓은 실제 데이타에서는 양자화 레벨이 낮은 PCR-트리(4)와 CR-트리(4)가 그림 13~15에서 보이는 것처럼 PR-트리의 성능보다 떨어진다. PCR-트리의 경우에는 두 기법을 결합하여 사용하기 때문에 CR-트리와 마찬가지로 부모 MBR의 크기에 영향을 받는다. 하지만, 확장되는 영역이 커짐에 따라 중복되는 값이 증가하게 된다. 즉, 데이타 영역의 크기가 클 경우 MBR 압축 기법으로 인해 단점을 가질 수 있지만 부분 MBR 기법을 통해 단점을 어느 정도 극복할 수 있다. 그림 10~12에서의 결과보다 CR-트리와 PCR-트리의 성능 차가 그림 13~15에서 더 큰 것을 볼 수 있다. 두 기법을 결합하여 사용하는 PCR-트리는 좌표의 중복, 데이타 영역의 크기, 그리고 압축 레벨에 영향을 받는다. 그럼에도 불구하고, PCR-트리(8)는 그림 10~15에서 가장 우수한 것을 볼 수 있다. 이것은 8비트의 양자화 레벨이 데이타의 크기에 영향을 가장 적게 받는 적당한 양자화 레벨이라고 볼 수 있다.

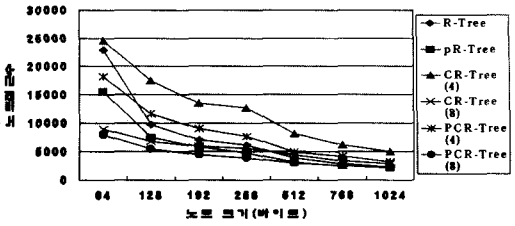


그림 13 노드 접근 수 (실제 데이타)

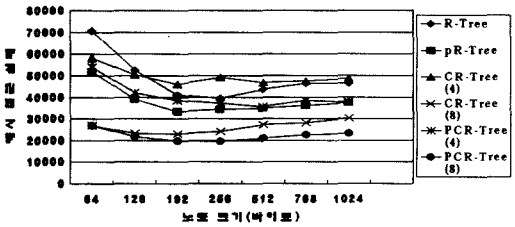


그림 14 캐시 접근 실패 수 (실제 데이타)

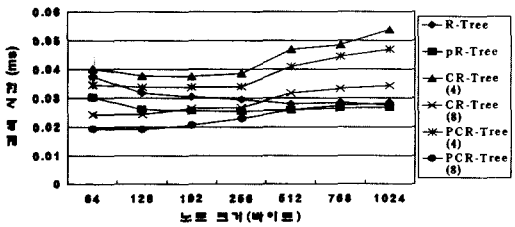


그림 15 검색 시간 (실제 데이타)

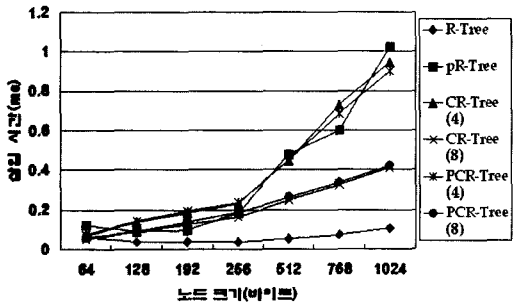


그림 16 삽입 시간

삽입 성능에 대해서는 엔트리를 하나씩 삽입하는 삽입 연산을 통해, 10,000개의 엔트리가 삽입되는 동안 걸리는 시간을 측정하였다. PCR-트리는 다른 색인 구조들 보다 삽입 알고리즘이 복잡하다. 상대적인 좌표를 계산하고 압축 기법을 사용하는 것이 이득인가를 확인하여, 그에 따라 노드를 구성해야 한다. 그렇지만, 그림 16을 보면 PCR-트리(8)가 R-트리를 제외한 색인 구조들에 비해 좋은 성능을 보인다. PCR(8)의 삽입 성능은 64바이트 일 때 가장 좋으며, 노드 크기가 64바이트와 128

바이트 일 때는 R-트리의 삽입 시간과 유사하다. PCR-트리에서 노드의 형태를 결정하기 위한 계산은 노드의 크기가 증가할수록 비용이 증가하여 R-트리의 삽입 시간보다 성능이 저하되는 것을 볼 수 있다. 하지만, 이러한 계산이 분할을 최소화하기 때문에 CR-트리와 유사한 성능을 보이고 있다.

그림 17은 랜덤한 10,000개의 엔트리를 삭제하는 동안 걸리는 시간을 측정한 결과를 보인다. 삭제 연산은 검색과 노드 갱신을 결합한 연산이다. 검색 시간에 대해서는 그림 12와 15에서 보이는 것처럼 PCR-트리(8)의 성능이 가장 우수하다. 하지만, 삭제될 엔트리로 인해 AMBR이 변경되면 노드를 재구성해야하고, 이러한 연산 비용은 노드 크기가 커질수록 증가하기 때문에 성능이 저하된다. 그렇기 때문에 그림 17에서처럼 노드 크기가 작을 때에는 PCR-트리(8)의 성능이 가장 우수하지만, 노드 크기가 증가할수록 R-트리의 삭제 시간이 더 뛰어난 것을 볼 수 있다.

그림 18은 벌크로딩 기법을 사용하여 색인 구조를 만들고, 삽입 연산을 통해 엔트리들을 삽입한 후에 측정한 검색 성능 결과이다. 4장의 시작부분에서 언급했었던 것과 같이 다차원 색인 구조의 검색 성능은 삽입 방법에 따라 다르다. PCR-트리(8)는 삽입한 엔트리 수에 상관없이 모든 경우에서 다른 색인 구조들보다 우수한 성능을 보인다. 그림 18에서 보인 것처럼 다른 색인 구조들은 삽입하는 엔트리 수가 증가함에 따라 접근 노드 수가 급격히 증가하는 반면에, PCR-트리(8)는 천천히 증가한다.

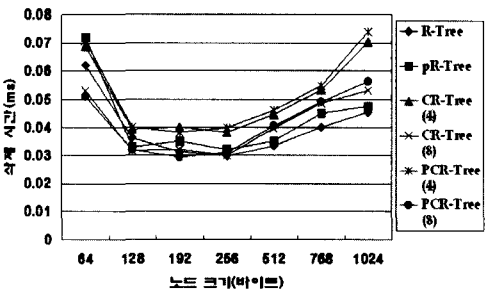


그림 17 삭제 시간

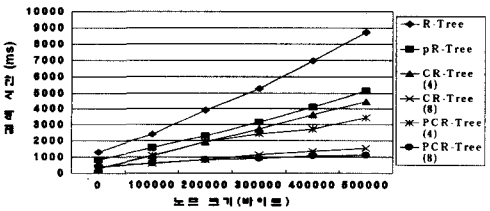


그림 18 삽입된 엔트리 수에 따른 검색 시간

5. 결 론

본 논문에서는 기존 방법의 특성들을 조합한 캐시를 고려한 다차원 색인 구조를 제안했다. 부분 MBR 기법과 MBR 압축 기법을 결합하여 엔트리의 크기와 캐시 접근 실패를 줄였다. 두 기법을 결합하여 구성한 노드가 CR-트리의 공간 효율보다 떨어지는 경우에는 MBR 압축 기법만을 사용함으로서, 두 기법을 조합하여 발생할 수 있는 단점을 보완하였다. 다양한 환경에서 폭넓은 성능 평가를 통해 제안하는 PCR-트리가 기존의 방법들보다 우수함을 보였다. PCR-트리(8)가 삽입 시간에 대해서는 기존의 색인 구조들과 유사한 결과를 보였으며, 검색에서는 모든 부분에 걸쳐 다른 색인 구조들에 비해 뛰어난 결과를 보였다. 본 논문에서는 기존의 캐시를 고려한 다차원 색인 구조들에 대한 자세한 분석을 통해 향상된 성능의 색인 구조를 제안하였고, 제안하는 색인 구조가 기존의 색인 구조들 보다 우수함을 보였다. 향후 연구에서는 압축 레벨에 따른 제안하는 색인 구조의 성능을 분석한다.

참 고 문 헌

[1] Anastassia Ailamaki, David J. DeWitt, Mark D Hill and David A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," Proceedings of VLDB, pp. 266-277, 1999.

[2] Stefan Manegold, Peter A. Boncz and Martin L. Kersten, "Optimizing database architecture for the new bottleneck: memory access," VLDB Journal Vol.9, No.3, pp. 231-246, 2000.

[3] Jun Rao and Kenneth A. Ross, "Cache Conscious Indexing for Decision-Suppoert in Main Memory," Proceedings of VLDB, pp. 78-79, 1999.

[4] Jun Rao and Kenneth A. Ross, "Making B+-Trees Cache Conscious in Main Memory," Proceedings of ACM SIGMOD, pp. 475-486, 2000.

[5] Philip Bohannon, Peter Mcllroy and Rajeev Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys," Proceedings of ACM SIGMOD, pp. 163-174, 2001.

[6] Shimin Chen, Phillip B. Gibbons and Todd C. Mowry, "Improving Index Performance through Prefetching," Proceedings of ACM SIGMOD, pp. 235-246, 2001.

[7] Kihong Kim, Sang K. Cha and Keunjoo Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," Proceeding of ACM SIGMOD, pp. 139-150, 2001.

[8] I. Sitzmann and P.J. Stuckey, "Compacting discriminator information for spatial trees," Proceedings of the Thirteenth Australasian Database, pp. 167-176, 2002.

[9] Guttman, A., "R-trees: a Dynamic Index Structure for Spatial Searching," Proceedings of ACM SIGMOD, pp. 47-47, 1984.
[10] Scott T. Leutenegger, J. M. Edgington, Mario A. Lopez, "STR: A Simple and Efficient Algorithm for R-Tree Packing," Proceedings of ICDE, pp. 497-506, 1997.
[11] Scott T. Leutenegger, "Multi Dimensional Data Sets," http://www.cs.du.edu/~leut/MultiDimData.html

심 정 민
2002년 충북대학교 정보통신공학과 졸업(공학사). 2004년 충북대학교 정보통신공학과 졸업(공학석사). 2004년~현재 한국전자통신연구원 재직 중. 관심분야는 데이타베이스 시스템, XML, 다차원 색인구조 등

민 영 수
1998년 충북대학교 정보통신공학과 졸업(공학사). 2001년 충북대학교 정보통신공학과 졸업(공학석사). 2001년~현재 충북대학교 정보통신공학과 박사과정. 관심분야는 데이타베이스 시스템, XML, 다차원 색인구조, 동시성 제어 및 회복 등

송 석 일
1998년 충북대학교 정보통신공학과(공학사). 2000년 충북대학교 정보통신공학과(공학석사). 2003년 충북대학교 정보통신공학과(공학박사). 2003년 3월~2003년 8월 KAIST 전자전산학과 박사후연구원 2003년 8월~현재 충주대학교 컴퓨터공학과. 관심분야는 데이타베이스 시스템, 트랜잭션, 저장 시스템, 멀티미디어 정보검색, XML, 정보검색 프로토콜 등

유 재 수
정보과학회논문지 : 데이타베이스
제 31 권 제 1 호 참조